GoLab

A Collaborative Go Playground using Replicated Growable Array CRDTs

Eduard Kagrimanyan (m9r8), Nathan Su(b0z8), Tim van der Kooi (b4n0b), Michael Yau (i5n8)

Introduction

For the past 10 years, distributed collaborative editors have given users the ability to share, access, and edit files simultaneously with other users all across the world. One of the most popular examples of a distributed collaborative editor is Google Docs, an excellent tool that allows teams to edit or read a shared text document concurrently. Although it appears that each user is editing the same file, each user has their own copy - or replica - of that file to make their changes. When these changes are distributed to all the other users, the system maintains consistency among all local replicas - that is, after having seen the same set of changes, the document appears to be the same for all users.

Ensuring data consistency across users is a challenge in distributed collaborative editors. Allowing users to make changes to their replicas simultaneously can result in replicas updating in an unpredictable manner (for example, due to network latency). Figure 1 below shows two scenarios of a simple replicated set with and without conflicts.

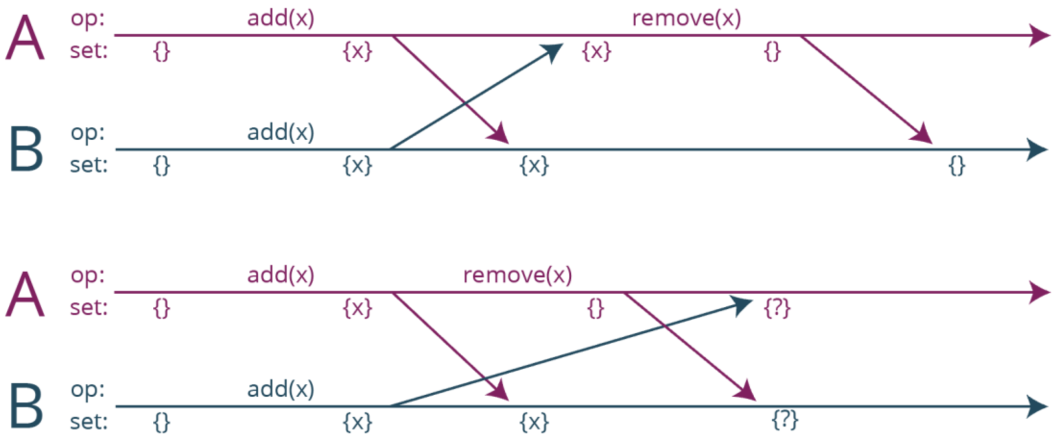

Figure 1: A naive approach for replicated sets

Due to delays inherently associated with communications over a network, strong consistency (where every user is always aware of every other user's current state at any point in time) is not possible. However, several approaches have been developed to maintain consistent replicated data. One such example is the conflict-free replicated data type (CRDT). Essentially, CRDTs can ensure strong eventual consistency, meaning that all users can make changes to their replicas concurrently, and all users who have seen the same set of changes, regardless of the order in which they were received, are guaranteed to share the same state. This is an example of operation-based CRDTs, where consistency is made possible by assigning each user a unique, comparable identifier and ensuring that operations follow three rules:

1. All operations (between users) must commute
2. The system must support exactly-once delivery semantics
3. Received operations must be applied in-order of creation

These points are illustrated in the figure below. The top half shows how an observed-remove set resolves conflicting add and remove operations. The lower half describes a situation which may occur if rule #3 above is not followed.

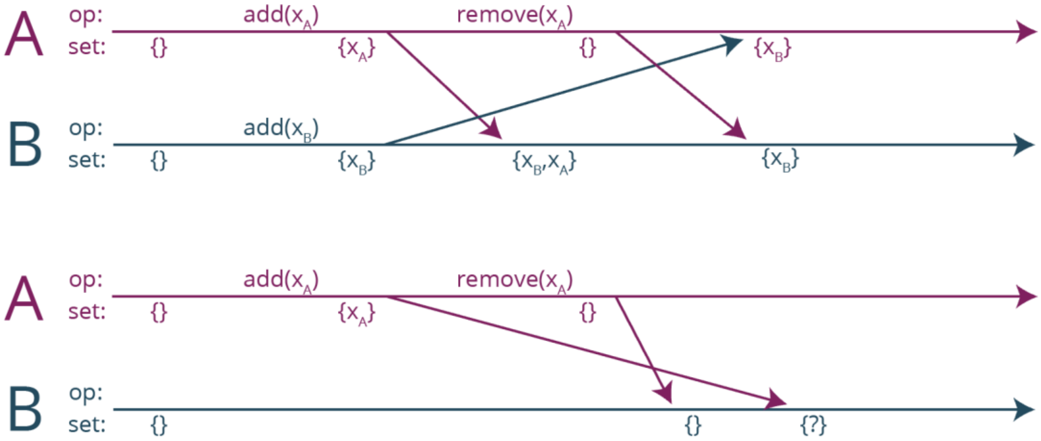

Figure 2: The observed-remove set CRDT

To increase our understanding about distributed collaborative editors and CRDTs, our project will focus on developing a collaborative code editor for the Go programming language where software developers can share code that they can edit, compile, and run within the application. The scope of our project is contained in the following key ideas:

1. We will create a browser application akin to The Go Playground (https://play.golang.org/), but with concurrently editable sessions.
2. We will use *replicated growable array* (RGA) CRDTs [6] to manage concurrent operations from users in our application, such as multiple developers adding and deleting sections of code at the same time.
3. We will implement a simple distributed file system supporting backup and retrieval of code snippets and execution logs.
4. We will implement a load balancer to distribute the workload of compiling and executing code to a network of connected machines, which will allow users to concurrently run their code on our application and collaborators to see the results.
5. The system will be deployed on Microsoft Azure.

System Overview

Our system is comprised of three logically distinct layers (Figure 3): multiple application instances, a network of workers, and a simple distributed file system for persistent storage.

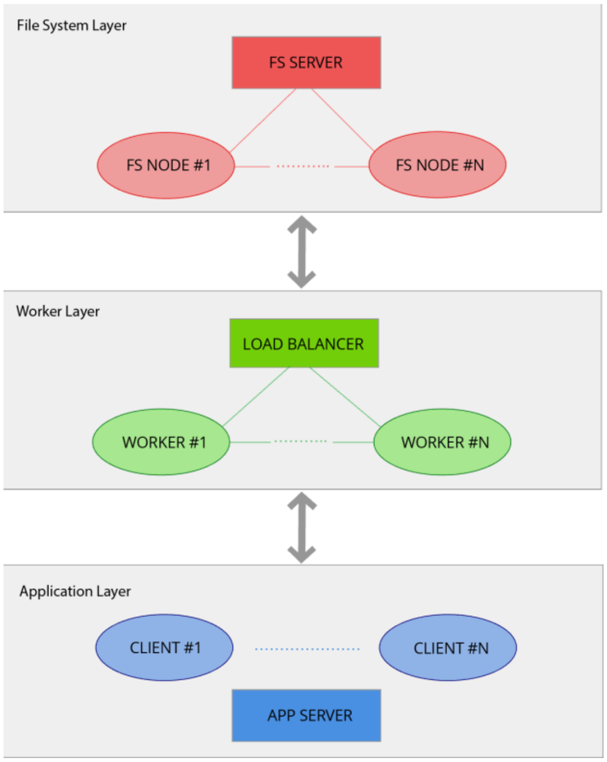

Figure 3: System components

Application Layer

Client

A client is a browser session connected as a user in the system. The browser session provides an interface for the user to edit, compile, and run code snippets with other users. Each newly created snippet generates a session ID which can be shared with collaborators who wish to edit that same snippet.

When a user connects to the application server (described below), they are presented with the application which receives the IP of the dedicated worker node (see section on Worker Layer below). The browser then establishes a WebSocket connection to the worker.

Application Server

The application server is responsible for bootstrapping a new client, which includes:

- Delivering necessary HTML and Javascript.
- Delivering the latest version of the code snippet (if given a session ID)
- Retrieving the IP of a worker node from the load balancer (see section on Worker Layer below) for a client.

Worker Layer

Worker Node

A worker node is responsible for disseminating operations, maintaining a local copy of any number of code snippets (sessions), and compiling and executing code snippets. Each worker has the following responsibilities:

- On receiving an operation from a client or worker, integrate the operation into the local sequence CRDT of the corresponding session and disseminate the operation to connected workers.
- Periodically save the code snippet to the distributed file system according to the local sequence CRDT, identified by its session ID.
- Maintain heartbeats to connected clients
- Compile and execute code snippets - a worker node can be selected to complete a job to compile any code snippet in the system if the load balancer selects it. The node will maintain a queue of jobs should it receive more than one job to execute. This is described in further detail in Code Compilation and Runtime Environment below.

Load Balancer

The load balancer is responsible for managing all the worker nodes in the worker layer. It has the following responsibilities:

- Balance client-to-worker load between all workers.
- Manage worker-to-worker connectivity by attempting to maintain some pre-defined network topology.
- Maintain a record of active workers using heartbeats.
- Managing run requests sent by workers in case of worker failure
- Disseminate execution logs to all connected workers

When a new worker joins the network, the load balancer must decide which other workers the connecting worker should connect to.

Assumptions: Given that the load balancer is essential for connecting clients and adds no value other than providing a gateway into the system, it is fair to assume it never crashes.

File System Layer

File System Server

The file system server is responsible for managing a simple distributed file system comprised of a network of file system nodes. It has the following responsibilities:

- Provide the IP of a connected file system node upon receiving a read or write request from a worker node.
- Maintain a directory of editable code snippets, encoded as sequence CRDTs, each identified by a session ID, stored on the distributed file system.
- Maintain a directory of all job requests, corresponding to saved code snippets ready to be compiled, each identified by a job ID, stored on the distributed file system.
- Maintain a directory of execution logs, corresponding to code snippet - execution log tuples, each identified by a job ID, stored on the distributed file system.

File System Node

The file system node is responsible for handling read and write operations. The file system nodes act as replicas to host, as well as receive read and write requests for sequence CRDTs, Go source files, and execution logs.

Collaborative Editing

There has been much work done in the area of collaborative editors, and over the years many techniques and approaches have arisen to handle collaboration. Of the many approaches: two main categories dominate: Operational Transformations (OT) and Conflict-Free Replicated Data Types (CRDT).

A good example of OT is Google Docs [7], which relies on a centralized server to handle merges (Figure 4). In this case, two separate nodes may make edits to the same position, but a centralized server handles and potentially modifies the operations to guarantee conflict-free merges.

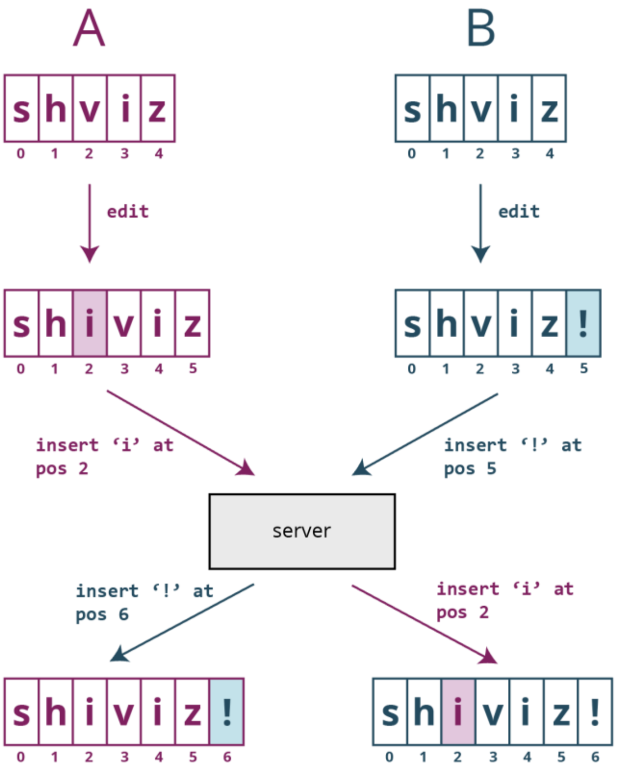

Figure 4: Operational transformation

On the other hand, CRDTs handle concurrency without the need of a centralized server, enabling the use of a peer-to-peer network (Figure 5). For this reason, we have decided to use CRDTs for this project, as it alleviates the bottlenecks and single points of failure associated with OT. More specifically, we have decided to use Replicated Growing Array (RGA) sequence CRDTs, where the CRDT is a sequence of elements containing a unique identifier, a position, and content for some character in a file. One potential issue to consider with RGA approach is the persistence of tombstones, which are sequence elements that have been previously "removed" through a remove operation but remain in the sequence as reference for future operations. The obvious problem with this is the impact on space complexity as many elements are removed and added, constantly growing the sequence. Nonetheless, this is not a significant problem in our project because the scope of GoLab is not intended to be extremely large scale.

Regardless of the collaborative approach chosen, the following three properties are fundamental:

1. **Causality**: operations exist in correct precedence order.
2. **Convergence** (Eventual Consistency): all local copies of the sequence CRDT across the system converge to identical copies when the system is idle.
3. **Intention**: every operation should have its expected effect across all local copies of the sequence CRDT in the system.

Ordered-list CRDTs using the RGA strategy store the sequence as a linked list [8]. Figure 5 has a simple sequence of nodes A and B inserting characters in different locations. The inserts are added with partial ordering by referencing the position before it. So when node A adds "i" in between "1a" and "2a", the operation will reference "1a" as the element before the new insert. The identifier for every character is unique and thus will achieve consistency between different nodes. We chose this approach because it allows for operations to be character-based as opposed to line-based, thus allowing concurrent same-line editing.

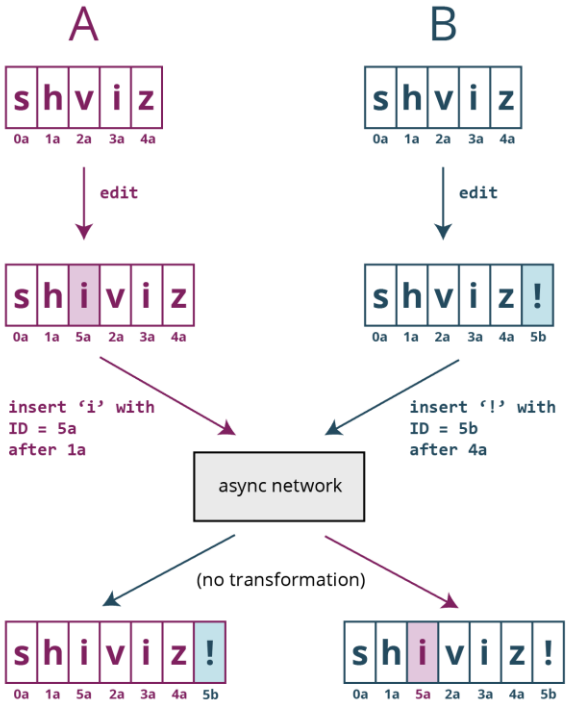

Figure 5: The ordered-list CRDT, editing at different locations

Figure 6 details the scenario where two different nodes insert a character into the same position in their sequence CRDT. When the operation is disseminated to the other nodes in the network, the nodes see that both inserts point to the same previous element. Since element identifiers are unique, the two conflicting elements' identifiers can be compared and the greater of the two will have precedence. This semantic ensures consistency when operations are disseminated. Deletions create a tombstone by setting a flag to hide the element in the textual rendering. Tombstones are needed in the sequence because of how insert elements reference previous elements. If we delete the element from the sequence, an operation from another node may reference that deleted element and thus will have no way of resolving it.

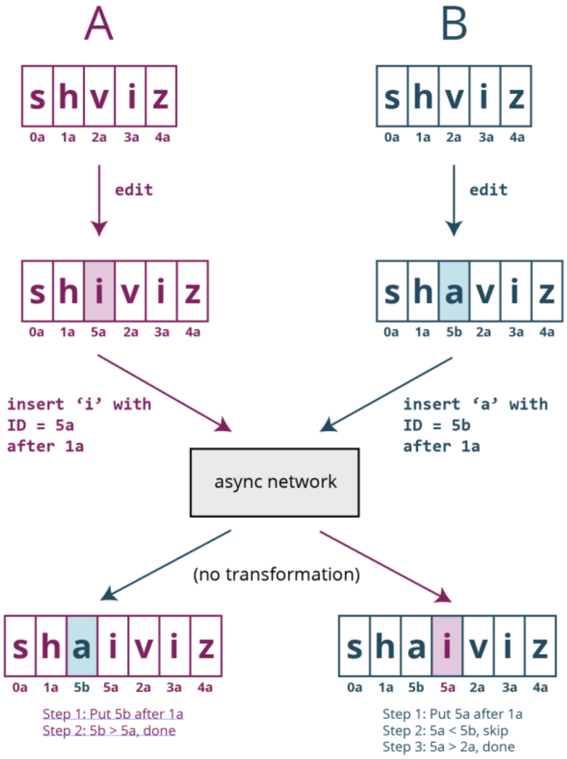

Figure 6: The ordered-list CRDT, editing at the same location

Operations

Operations correspond to an add or a remove of a character in a snippet, where each operation has a unique id and position reference. Referring to figure 7 and 8, as the user modifies a snippet, new operations are created and added to a local array of recent operations performed by the user. Every few seconds, this local array of operations is sent to the clients' worker node, who is responsible for disseminating the operation to other works. Furthermore, the worker sends its own array of operations (accumulated through incoming operations) to its connected clients.

When a client receives operations from the worker, it must update its local CRDT with the received operations. This means that all CRDT handling is done on the client only, leaving operation dissemination to the workers. The reason we chose to do this is because the the browser (JS) would already have to understand how operations are being performed and it would be very difficult to determine where in the sequence an operation is applied. So the most feasible option was to leave it to the client to maintain local file state.

As the user edits the file, the client will periodically push its current snippet in text format to its worker, which will subsequently save the snippet to the file system.

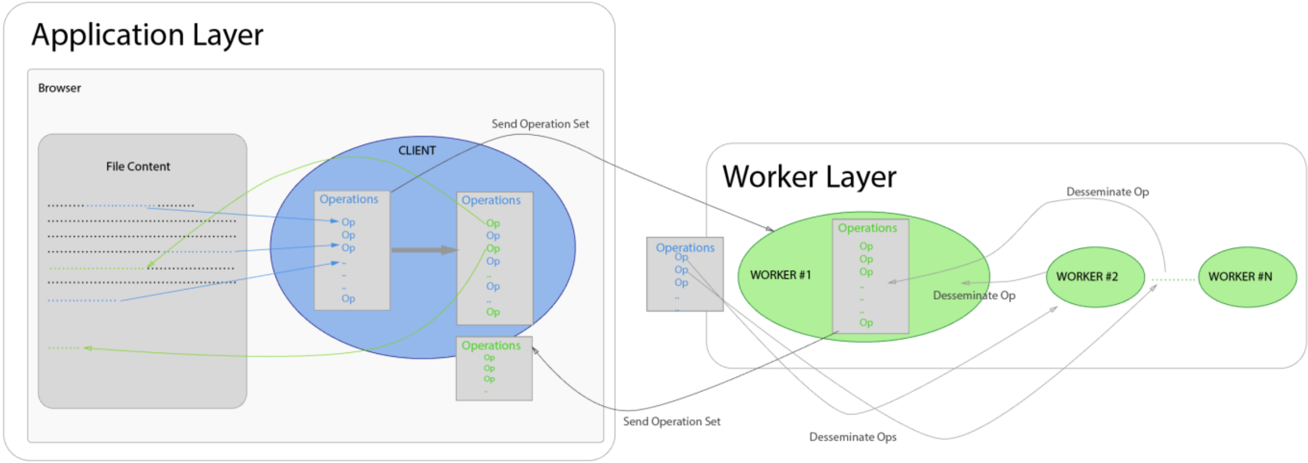

Figure 7: Client-worker operation exchange: as the user updates the snippet, operations are added to its local array and as operations are disseminated to works they are added to the worker's local array. Both client and worker periodically exchange operations to sync all snippet changes.

Code Compilation & Runtime Environment

Protocol

When users are ready to run their code, our application goes through a protocol that involves each layer of our system architecture. This protocol is detailed in a sequence diagram (Figure 8) and list below:

- Notifying the file server and the load balancer:
    - First, the user will press the run button in the application, which notifies its connected worker node.
    - The worker node will call the file server to save the client's code snippet as a pending job, and receives a job ID to use for the compilation of that code.
    - The worker node passes the job ID to the load balancer, which uses a consistent hashing strategy to distribute jobs fairly across the worker node network.
- Finding a worker node and executing the code:
    - The load balancer will pass the job ID to a free worker node to compile and run the code (if none are free, pick the node with the least amount of jobs in its job queue).
    - The selected worker node will call the file server with a job ID to retrieve the code snippet to compile - or if the code has already been compiled and executed previously, it will receive the execution log for that compiled code.
    - The worker node will save and run the code on its machine locally. If the code does not compile, any error(s) returned will be reflected in the log.
- Distributing the results:
    - Once the code is compiled, the worker node will save the log output on the file server with the job ID.
    - It will also send the log to the load balancer to broadcast to all connected worker nodes, which will then send the log to all of their connected clients.

Failure Cases

Throughout this protocol, there are several points of failure that will be handled appropriately by our system. In the sequence diagram in Figure 8, there are four major regions of failure that are marked with the numbers 1 to 4. Those error cases will be handled as such:

1. If Worker A fails at any time before sending an ok() back to Client A, that client will reconnect to a new worker node (via the load balancer) and resend the run() command to that new node. Client A will know if Worker A fails because they will utilize a heartbeat mechanism to assert connection status.
2. If Worker A fails after sending an ok() back to Client A, that client will reconnect to a new worker node but will not resend the run() command to the new node. It will not resend the command since the load balancer will find some worker node to complete the job, and will also distribute the results of that job to every worker node. The client will eventually receive the execution log.
3. If Worker B fails any time between receiving the run() request and sending the final log output to the file server, the load balancer will find a new worker node and call run() on that node.
4. If Worker B compiles and runs the code, saves the log output to the file server, but fails to send log output back to the load balancer, the load balancer will assume the worker has failed. It will then ask a new worker node to run the code snippet, but that worker node will find out that the snippet has already been executed when it receives a log from the file server instead of the snippet source. It can then proceed to return the log to the load balancer.

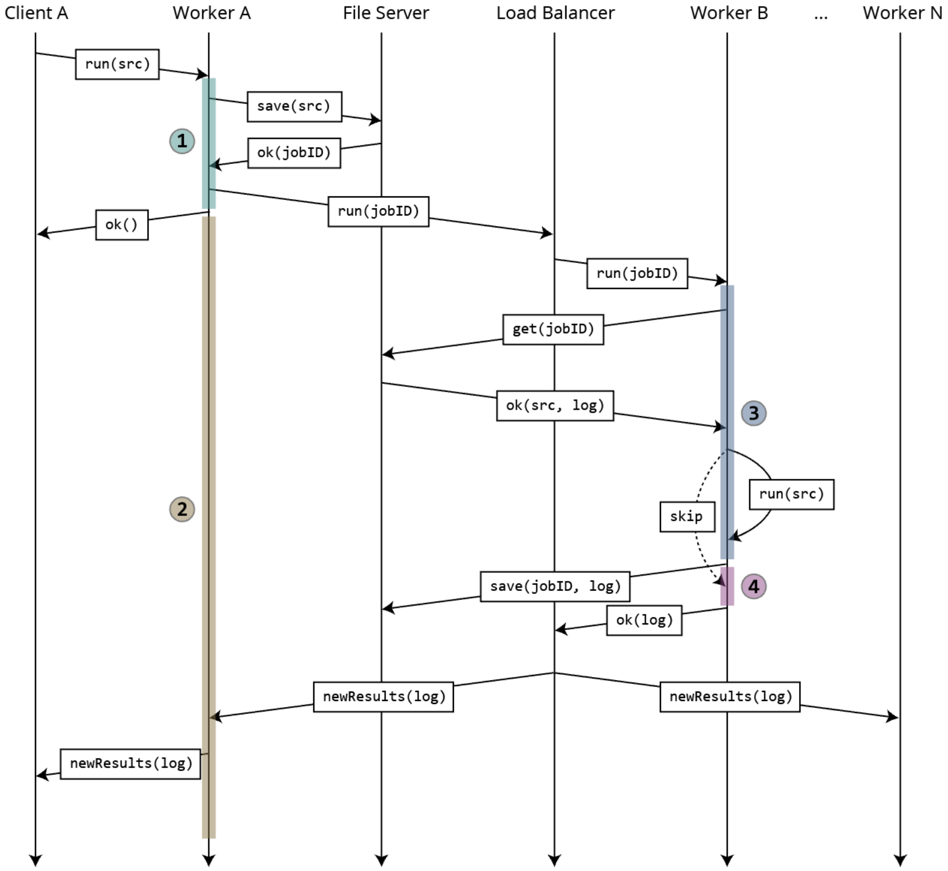

Figure 8: Sequence diagram for code compilation protocol with numbers indicating potential failure cases

Assumptions

Assumptions made:

- Nodes in the system can be trusted, and do not attempt to compile and execute potentially malicious code
- The app server, load balancer, and file system server are always online, never fail, and never misbehave
- File system nodes propagate write requests to neighboring nodes through the file system server and all nodes act as replicas for the same data. The file system server maintains consistency.
- A timeout of 2s is used to detect failed worker and file system nodes.
- A node that has failed may or may not ever reconnect
- A maximum of 16 worker nodes, 16 file system nodes are in the system
- Each session has a maximum of 4 concurrent users
- A maximum of 64 users are connected to the system at any given time

Assumptions not made:

- Nodes are synchronized
- Nodes communicate failures
- Code snippets compile

Testing

We will employ three testing methods to ensure our distributed system and application are benchmarked to our expectations. For all of the below test categories, we will test locally and on Microsoft Azure.

In addition to the above, it is also necessary to ensure these properties are upheld in the file copies maintained in the file system.

Unit Tests

Unit tests will be written to ensure API's on each node type behave correctly. Since the behaviour of these API's largely depend on having active nodes running to establish communication, we will have to mock some behaviour. With that said, it is best to leave simple test cases to the unit tests and the more complicated behaviour to the integration tests.

Integration Testing

Integration tests will be written to ensure correct behaviour between active nodes, which includes API testing. To achieve this, we will have a main script that can bootstrap the entire systems topology and kick off test cases using the objects available in the main script. This can include nodes joining and leaving the system, nodes failing intermittently, nodes communicating, etc.

Because the integration tests are all running within a single script, concurrency, timing and failures will be very easy (and fairly consistent to test). For instance, supposing we have two worker nodes, we can have both of these nodes issue conflicting operations at the same time.

Manual Testing

Since the core of the system tests will reside in the unit and integration tests, the remaining testing necessary is that of the UI and application. This would include multiple people using the application at the same time and ensuring that the UI works as expected.

Timeline

Date	Task
March 12	[Team] Complete API specs for all communication layers
March 14	[Tim/Michael] Basic server APIs implemented for each communication layer
March 16	[Tim/Michael] Basic worker/file system node APIs implemented
March 19	[Eddie/Nathan] Basic UI/client node implementation completed [Tim/Michael] Worker/file system nodes can connect to servers/each other
March 21	[Tim/Michael] Ideal code compilation protocol works (not including failures)
March 23	[Team] Email/meet with TA to update project status [Team] Iterate over current progress; reassess what needs to be done [Eddie/Nathan] Basic RGA CRDT algorithm in place
March 26	[Eddie/Nathan] UI polished and client connected to backend systems [Tim/Michael] Code compilation protocol can handle failures
March 28	[Eddie/Nathan] RGA CRDT working with user input across clients
March 30	[Team] Ramp up integration testing and start working on bug fixes
April 2	[Team] Finish up outstanding features/eliminate as many bugs as possible
April 4	[Team] Code freeze/final testing - spend until deadline testing cases on Azure
April 6	Project code and final report due
April 9-20	Prepare for demo (date TBD)

SWOT

Strengths

- [General] Experience with distributed file system (A2)
- [General] Experience with basic client-server distributed systems (A2)
- [General] Quickly adapt to sleep deprivation and anti-social lifestyle
- [General] Team members have co-op experience in both front- and back-end development
- [Eddie] Expertise with scripting system bootstrapping for automated test execution (A2 and Project 1)
- [Nathan] Expertise with Azure (Project 1)
- [Tim] Expertise with Dev Ops, specifically Jenkins and other CI tools, which can serve as a good asset deciding and understanding UI features beyond simple code editing

Weaknesses

- [General] No CRDT experience: despite having many resources to help us understanding CRDT and related algorithms there is a learning curve involved.
- [General] Limited API development experience: there may be a learning curve associated with this as well, since API schemas and semantics will not come naturally.

Opportunities

- CRDTs are well researched, such that there are plenty of good resources for us to reference our implementation.
- Collaborative editors like Google Docs are common, which provide good reference implementations.
- We may leverage GoLab itself to write some of GoLab! This provides a good opportunity to showcase GoLab's abilities.
- The file system we are implementing is very simple: given the simplicity of the FS and our experience from A2, we can shift most of our focus to all the other aspects of the project.
- [General] We are motivated to be a formidable match to the other editors being developed for Project 2

Threats

- Scope: we are committing to a lot of work, which could threaten successful execution of what we are committing ourselves to.
- Course commitments: most of us are taking 4+ courses, which reduces the amount of time dedicated to the project.
- Time: due to the above concerns with scope and commitments, 4 weeks will be a very tight squeeze.

References

[1] https://hal.inria.fr/inria-00336191/PDF/main.pdf

[2] https://hal.archives-ouvertes.fr/file/index/docid/921633/filename/fp025-nedelec.pdf

[3] https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type

[4] https://www.youtube.com/watch?v=9xFfOhasiOE

[5] https://www.microsoft.com/en-us/research/video/strong-eventual-consistency-and-conflict-free-replicated-data-types/

[6] https://martin.kleppmann.com/2016/11/03/code-mesh.html

[7] https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html

[8] http://simongui.github.io/distributed-systems/crdt.html

[9] https://github.com/netopyr/wurmloch-crdt